Extended features and evaluation of aggregating OPC UA servers

Karri Kumara

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 17.5.2017

Thesis supervisor and advisor:

Ilkka Seilonen, D.Sc. (Tech.)

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Karri Kumara

Title: Extended features and evaluation of aggregating OPC UA servers

Date: 17.5.2017 Language: English Number of pages: 7+37

Department of Electrical Engineering and Automation

Major: Information and Computer Systems in Automation

Supervisor and advisor: Ilkka Seilonen, D.Sc. (Tech.)

OPC United Architecture (OPC UA) is a protocol for data exchange that is based on a server-client model. OPC UA servers are used in industry to collect and organize data from devices. An aggregating OPC UA server collects data from one or more source OPC UA servers and its client serves as the interface to the thus created OPC UA server network. OPC UA has some advantages to its competitors, including built-in support for Historical Access (HA) and Alarms And Conditions (A&C). HA allows storing and reading previous data values, and A&C allows informing external systems of the states of the server.

This thesis defined requirements for incorporating HA and A&C to an aggregating OPC UA server and designed how this would be done using software design principles. The requirements were based on a study of the OPC UA specification and other relevant literature. The A&C and HA cases chosen to be designed were based on the aggregating server relaying received requests to source servers. Extensions based on the designs were implemented to a prototype aggregating OPC UA server that was run through tests that evaluate its performance both during configuration and runtime, based on simple use cases. The Prosys Java OPC UA Software Development Kit was used to create and run the server and the tests.

The implementation of request relaying-based A&C and HA on an aggregating OPC UA server was found to be simple. This thesis found the performance of the aggregating server to be adequate for simple use cases, but possibly inadequate for multilayer server networks or contexts requiring near real time responses. The performance evaluation of subscriptions, diversified address space transformations and improved configuration times were identified as potential targets for further development.

Keywords: OPC UA, aggregating server, history data, alarms and conditions

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Karri Kumara

Työn nimi: Aggregoivien OPC UA-palvelimien laajennetut ominaisuudet ja arviointi

Päivämäärä: 17.5.2017 Kieli: Englanti Sivumäärä: 7+37

Sähkötekniikan ja automaation laitos

Professuuri: Automaation tietotekniikka

Työn valvoja: TkT Ilkka Seilonen

Työn ohjaaja: TkT Ilkka Seilonen

OPC Unified Architecture (OPC UA) on tiedonsiirtoprotokolla, joka perustuu asiakas-palvelin-malliin. OPC UA-palvelimia käytetään teollisuudessa datan keräämiseen laitteista ja sen organisointiin. Aggregoiva OPC UA-palvelin kerää dataa yhdeltä tai useammalta OPC UA lähdepalvelimelta ja sen asiakas toimii rajapintana näin luodulle OPC UA-palvelinverkolle. OPC UA:lla on joitain etuja kilpailijoihinsa nähden, kuten sisäänrakennettu tuki Alarms And Conditionsille (A&C) ja Historical Accessille (HA). A&C mahdollistaa palvelimen tilojen viestimisen ulkoisille järjestelmille ja HA mahdollistaa aikaisempien arvojen tallentamisen ja lukemisen.

Tämä työ määritteli, miten aggregoiva OPC UA palvelin voi sisällyttää HA:n ja A&C:n, ja suunnitteli, miten tämä tehtäisiin käyttäen ohjelmistosuunnittelun menetelmiä. Vaatimukset pohjautuivat tutkimukseen OPC UA-spesifikaatiosta ja muusta relevantista kirjallisuudesta. Suunniteltavaksi valitut A&C:n ja HA:n tapaukset perustuivat pyyntöjen välittämittämiseen aggregoivilta palvelimilta eteenpäin lähdepalvelimille. Näiden suunnitelmien pohjalta toteutettiin laajennukset OPC UA-palvelinprototyyppiin, jolle ajettiin sekä konfiguroinnin- että ajonaikaista suorituskykyä arvioivia testejä perustuen yksinkertaisiin käyttötapauksiin. Prosys Java OPC UA Software Development Kitiä käytettiin sekä palvelimen että testien luomiseen ja ajoon.

Pyyntöjen välitykseen pohjautuvien A&C:n ja HA:n toteutus aggregoivalla OPC UA-palvelimella havaittiin yksinkertaiseksi. Aggregoivan palvelimen suorituskyky yksinkertaisissa käyttötapauksissa todettiin riittäväksi, mutta mahdollisesti riittämättömäksi usean kerroksen palvelinverkkoihin tai lähes reaaliaikaisia vastauksia vaativiin tilanteisiin. Mahdollisiksi lisätutkimusaiheiksi tunnistettiin subskription suorituskyvyn arviointi, monipuolisemmat osoiteavaruusmuunnokset ja konfiguraatio-aikojen parannukset.

Avainsanat: OPC UA, aggregoiva palvelin, historiadata, hälytykset

Preface

This thesis has been a long time in the making and faced several difficulties. I want to thank my supervisor Ilkka Seilonen and the people close to me for being patient with both the thesis and me. I appreciate all of your help, even when it led to disagreements. I believe it has been for the best.

I also want to thank my predecessors Tomi Tuovinen and Joona Elovaara for laying in the groundwork for the aggregating server used in this work. Your earlier work and documentation made this project possible.

One last thanks goes to the customer support staff at Prosys for responding to my questions about the OPC UA Java Software Development Kit.

Now, full steam ahead to the future!

Otaniemi, 17.5.2017

Karri P. Kumara

Contents

Symbols and abbreviations

Abbreviations

OPC UA	OPC Unified Architecture
SCADA	Supervisory Control And Data Acquisition
HMI	Human-Machine Interface
COM	Component Object Model
DCOM	Distributed Component Object Model
DA	Data Access
HA	Historical Access
A&C	Alarms & Conditions
UML	Unified Modelling Language
API	Application Programming Interface
MTL	Model Transformation Language

1 Introduction

1.1 Background

Collecting, organizing and analyzing data are important tasks especially for automation devices. Data gathered from automation devices needs to be accessed in a form that is comprehensible to both software and to human users. Thus, vertically integrating different levels of automation networks has gathered a lot of interest and demand. One way to achieve this integration is using gateways [1]. Automation gateways are designed to take data from automation devices and represent and allow managing the data in a software environment.

OPC Unified Architecture, commonly called just OPC UA, is an attempt at creating such a platform [2]. Like its predecessor, OPC Classic, it is a specification of a communication framework based on a client-server architecture. Servers serve as a gateway by hosting and organizing the data from devices, while clients are used to access and interact with the servers [1]. While OPC Classic is still popular, its long use over the years has exposed some of its weaknesses. OPC UA made several additions and improvements to the original, such as improved modelling capabilities and support for non-Windows platforms [3]. It is used primarily in industrial settings like factories to gather data such as sensor readings and internal device data values.

In turn, aggregating OPC UA servers are used for collecting data from other OPC UA servers much like they collect device data [4]. Aggregating OPC UA servers themselves are still OPC UA servers and can be aggregated further, potentially creating a pyramid-like structure of OPC UA servers and clients. The client of the top-layer aggregating server is typically the main interface of the servers for external systems and thus well thought-out organization of data on the top-layer aggregating server is important for ease of access.

OPC UA servers use a standardized data representation, allowing any OPC UA client to connect to the server and access the data without additional preparation [3]. Each server can determine its way of organizing the data it gathers. In the process of data aggregation from underlying servers, the data might need to be filtered and transformed. This process is called address space transformation. Address space transformation is important for efficient use of aggregating servers in order to represent the large amount of gathered data in an understandable way.

Two features of OPC UA are focused on in this work: Alarms and Conditions (A&C), and Historical Access (HA). A&C allows the server to inform its clients of events in the system. [5]. It is often important to be able to gather and pass on alarms and condition data from lower-level servers to the aggregating server because the server network is accessed through the client of the aggregating server. HA is used to collect logs of previous values and events on the server [6]. While a server that can access its current values is useful, the addition of HA allows better analysis of the performance of the devices e.g. through graphs and histograms.

Performance evaluations of OPC UA servers have been done previously, for example with a server wirelessly integrating agricultural devices such as tractors[7]. There is less research on the performance of aggregating OPC UA servers, and even

less of ones using HA or A&C.

Previous research in Aalto University [8] created an aggregating OPC UA server and its address space transformation algorithm using the Prosys [9] Java Software Development Kit. These are the starting point of this work.

1.2 Objectives

The research problem of this thesis is designing the addition of Alarms And Conditions (A&C) Historical Access (HA) features to a previously created aggregating OPC UA server, as well as the performance evaluation of a prototype server based on those designs. Currently, no existing designs for A&C on aggregating servers are available. For HA, some existing designs like the OPC UA Historian [10] are available, but they are not used as the basis for the requirements or design of HA in this thesis.

The designs of both A&C and HA will incorporate each required feature as defined by the OPC UA specification [2], such as services and related methods. The designs are used as basis for implementing extensions to the previously designed aggregating server prototype[8]. This prototype is run through tests that evaluate its performance. The aim of these tests is determining the efficiency of the transformation algorithm and if the algorithm is affected by increasing the size of the modeled system. In addition, they are used for analyzing how well the aggregating server performs during runtime in comparison to the source servers it aggregates.

1.3 Research Methods

This thesis studies the OPC UA specification, the existing literature and research done on the subject of OPC UA and especially aggregating servers. These are used as the basis for defining the requirements for aggregating servers using Historical Access (HA) and Alarms and Conditions (A&C).

The Java programming language and the Prosys OPC UA SDK for Java [9] are used to create and test the OPC UA aggregating server prototype. The starting point of the prototype will be the aggregating OPC UA server and transformation algorithm previously used in Aalto University [8], but these will be modified for the needs of this work.

The performance of the prototype is evaluated using tests using Java and the Prosys OPC UA SDK for Java. The tests will measure durations taken by different actions such as reading data and creating the aggregating server address space. For storing, analyzing and graphically presenting the testing results, LibreOffice Calc[11] is used.

This thesis uses charts to visually demonstrate design concepts, as well as Unified Modelling Language [12] sequence diagrams to demonstrate processes in the design.

1.4 Outline of the thesis

Chapter 2 examines OPC Unified Architecture through a literature study. It deals with the history of OPC UA then continues to summarizing the address space model,

as well as the Historical Access and Alarms And Conditions features of OPC UA.

Chapter 3 is the second part of the literature study, concerning the concept of OPC UA aggregating servers and explaining how and where they have been used, as well as with the address space transformation process. The chapter also details the aggregating server and transformation algorithm that serve as the basis of this work.

Chapter 4 discusses the Alarms And Conditions part of the OPC UA specification in the context of an aggregating server. The requirements and design of an aggregating server with A&C are detailed.

Chapter 5 defines the requirements of an OPC UA aggregating server using Historical Access, and details a design based on those requirements..

Chapter 6 uses graphs to present the results of the performance evaluation done on an aggregating server prototype based on the designs created in chapters 4 and 5. The testing environment and process are also described.

Chapter 7 contains a summary of what has been discussed in this work, conclusions based on the test results, as well as suggestions for extensions for the aggregating server and further research.

2 OPC Unified Architecture

Both OPC UA and OPC Classic are communication frameworks [2]. They are useful for collecting and aggregating data from various devices, which has made them popular for arranging data collection and inter-device communication for automation. They can be used to gather information from devices such as sensors, or control them through alteration of operating variables. This allow OPC to be used for complete control of factories and other such facilities. They are often used to focus device control and the presentation of data for human users, e.g. in SCADA and Human Machine Interface (HMI) implementations. OPC or OPC UA Servers collect data from devices and other servers and present it to clients, allowing them to read and make changes to the data. Different versions of OPC do this slightly differently.

The original OPC, also known as OPC Classic, uses Microsoft's Communication Object Protocol (COM) and Distributed Communication Object Protocol (DCOM) as the basis of its communication[3]. It also requires the use of the Microsoft Windows operating system. One of the first OPC specifications and the most commonly used is OPC Data Access, which allows reading and writing data. Later on, additional specifications were created, including Historical Access and Alarms And Events. While very popular, due to its dependance on Microsoft-specific technologies, it is not suited for the needs of every company. Also, due to its age and how it has been expanded over time with additional standards, it has some problems. These include poor integration of some of its extensions and somewhat limited capabilities for data modeling.

To create a possible solution for these issues, the OPC Foundation began development of OPC UA [3] . In comparison to OPC Classic, OPC UA is not restricted to the Windows platform and can thus be made to work on many kinds of platforms. It was designed from the beginning to include the functionalities of the OPC extensions and have them work together in a unified address space.

While its first version was released in 2006, implementations of OPC UA are still being actively developed on multiple programming languages, such as Java and C++. In addition to the OPC Foundation, companies such as Prosys [9], [13] , Softing [14] and Unified Automation [15] are developing their own software development kits for OPC UA.

2.1 Address Space Model

The address space of an OPC UA server means all the data on that server, presented to the clients according to the address space information model [3]. The address space has nodes, which are blocks of data on the server, and references, which are the relations between those nodes. Together, these form a graph. The exact structure of the node-reference network of the address space is not defined by the specification and should be defined based on the requirements of the application.

The OPC UA address space model is a meta model: It models how information is itself modelled in the address space [3]. It establishes node classes and base types that are used in the address space, which serve as the basis for more specific models.

Nodes can have several attributes and values as well references with other nodes [16]. The actual data of the system is stored in the attributes of nodes. Each node in an OPC UA address space has a unique NodeId attribute which is used to identify the node. The node also typically has other attributes, based on its type. A node can be one of several NodeClasses. The NodeClass of a node defines what attributes and references the node should have. There are seven different NodeClasses: Object, ObjectType, Variable, VariableType, Method, MethodType or View.

Objects do not have value attributes themselves, but are used to contain Variables and Methods [16]. They can represent physical or non-physical elements of the system. They are the primary way of organizing the address space, since they can contain other types of nodes. One type of object is the folder, which is a tool for organizing other nodes, much like a computer folder.

Variables contain values, which have a DataType such as integer or boolean. A simple variable simply contains one data value as an attribute, but a complex one might have any number of child variables, each with their own values and possible child Variables of their own, which could themselves be complex [16] .

A reference is always between two nodes and can be one- or two-directional [16] . The node containing the reference is called the source node and the referenced one is called the target. The ReferenceType of a reference is used to define the kind of relation that the two nodes have. For example, the type Organizes is used to portray a hierarchical relation between two nodes. If the relation is hierarchical, that reference cannot lead to a cycle. The nodes connected by a reference are not always in the same address space. In the case of this kind of remote reference, the remote node is identified with the name of the server and the name of the node on that server.

The type-nodes are used to make a new type definition, setting requirements for future node instances of that particular type [16] . For example, the VariableType ”CounterType” could define a kind of variable that is used as an integer counter going from 0 to some maximum value. The user can then instantiate a variable directly as a CounterType without needing to create the functionality from scratch. The CounterType in this example is a TypeDefinitionNode, linked to the node with the HasTypeDefinition Reference. The TypeDefinitionNode contains the actual metadata about what attributes a node of the presented type should have. TypeDefinitionNodes can also inherit from supertypes using the HasSubType reference. Inheritance works

similarly to inheritance in programming languages, e.g. Java. Subtypes inherit the characteristics of the supertype and nodes of the subtype can be used where the supertype is expected.

A subtype of objects, events are the occurrences such as data value changes and errors within the address space, which generate event notifications [16]. For clients to be informed of event notifications, they have to subscribe to an EventNotifier node on the server. An example of an EventNotifier would be one that provides events related to a particular sensor, such as alarms and value changes. Subscription and the special Event subtypes Alarms and Conditions are explained in greater detail later.

A Method is a software function that, as previously mentioned, is a part of the address space model and is always a component of an Object [17]. They are called to execute some procedure, can be given additional arguments for that purpose and thus resemble functions in programming languages like C. Methods are typically used for simple operations such as calculations. For longer, more ongoing operations on the server, programs are used. Programs in OPC UA are derived from finite state machines and run continuously, changing from state to state as necessary.

Figure 1 depicts an OPC UA server and shows how the address space relates to other parts of the server. The server itself has its own subcomponents [2]. The address space typically models some external objects, such as industrial devices. Also seen in the address space is a view, which is a subset of the address space that has been limited according to some specific interest [16], such as a subset containing only the alarms for monitoring purposes. The server Application Programming Interface (API) manages the address space and accesses the data in it. The communication stack is responsible for communication between the server and the OPC UA client. The client is used as an interface to the server by other servers or software, or human users.

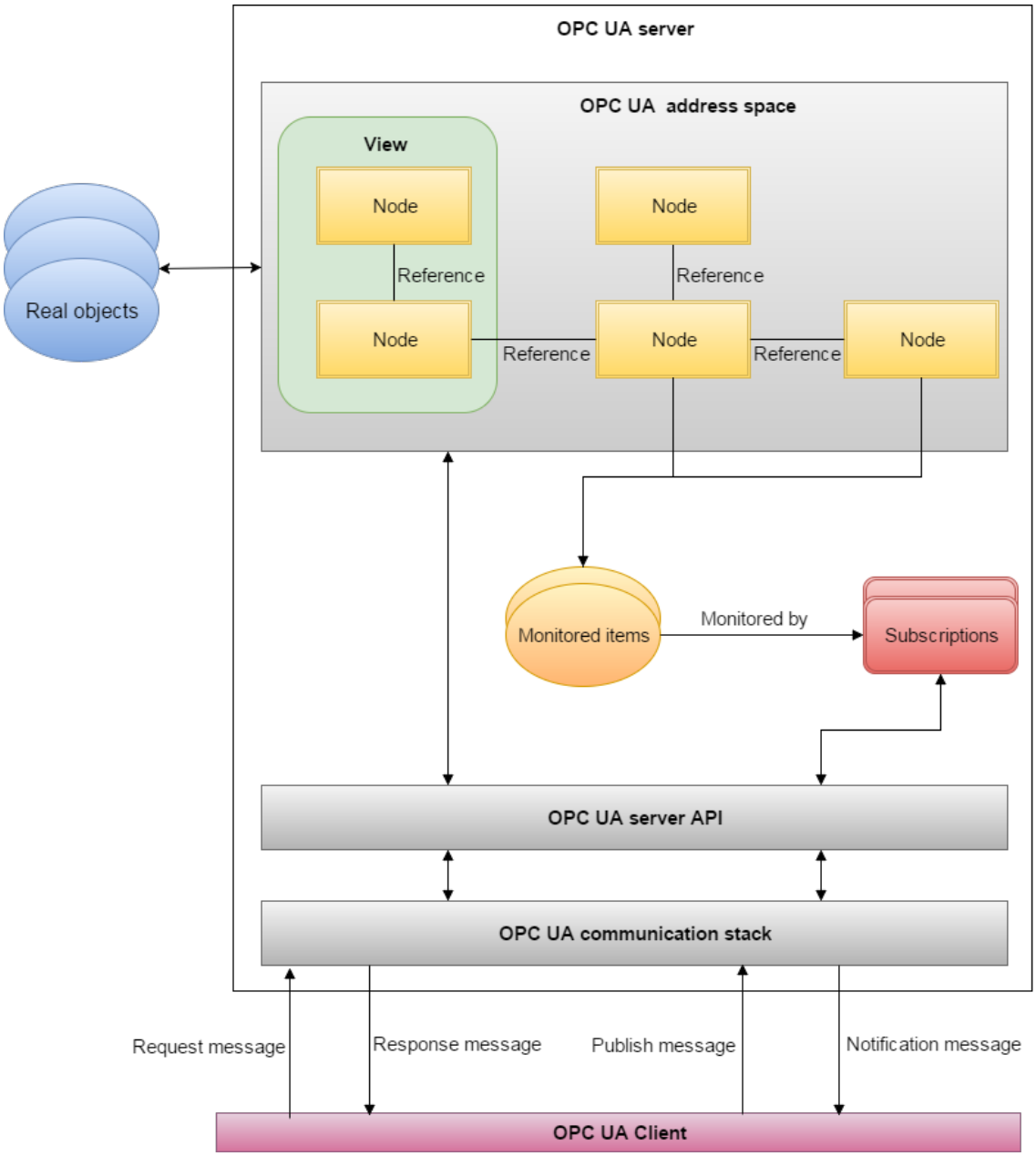

Figure 1: The structure of an OPC UA server [2].

Request messages, such as for Read and Write, are sent from the OPC UA client to the server and processed in the OPC UA server API. The API accesses the address space and retrieves or modifies some of its contents, and sends a response message to the client. Monitored items and subscriptions are a way for the client to continuously keep track of values in the address space.

2.2 Subscription

A subscription is a way for a server to automatically transmit data changes and event notifications between server and client [3]. There are three different items that can be monitored with a subscription: Variable values, events and aggregate values. In this case, aggregate values refer to values derived from raw data in some fashion, such as averages, minimums and maximums over a given time period [18].

Each subscription has a unique SubscriptionId used to identify it, as well as a processing priority relative to the other subscriptions created by the same client. PublishInterval determines how often the server sends forth queued notifications of changes to the client. A shorter publishing interval will allow observation of the server to be closer to real-time, but can be demanding for the hardware to process. PublishEnabled determines simply whether the notifications get delivered to the Client at all. If set to false, the monitored item can still generate notifications, but they are not published.

Each subscription can have multiple monitored items. Monitored items also have important subscription-related parameters, which can be seen in figure 2.

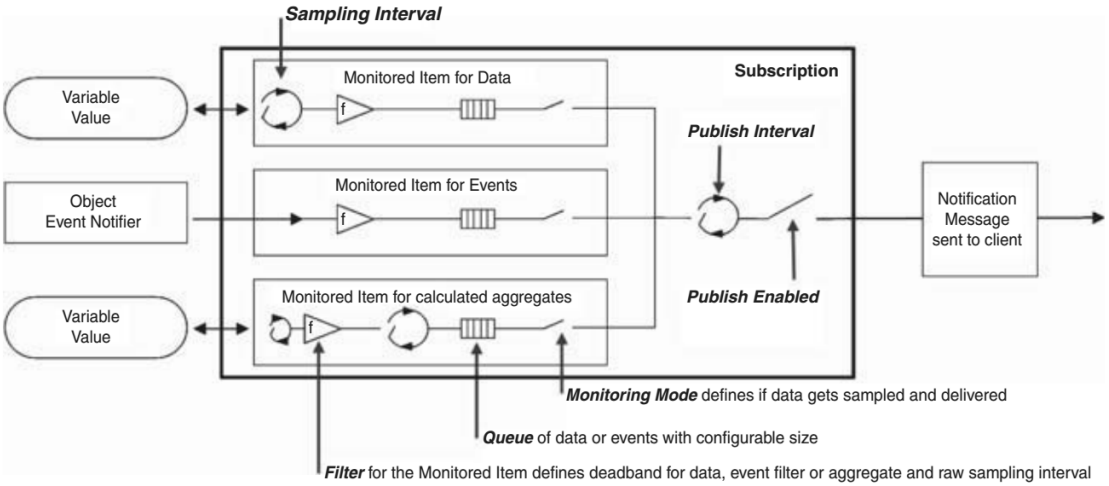

Figure 2: Subscriptions and monitored items. [3]

MonitoringMode determines whether the monitoring for the item is currently active or not. Turning monitoring off for one item does not affect the monitoring of the other items in the subscription. SamplingInterval determines how often the monitored item is checked for changes, or how often an aggregate value is calculated. This setting depends heavily on the monitored item. The level measurement of a liquid tank could be checked updated often, but something like an outside temperature measurement might only need to be updated once per hour. QueueSize determines how many notifications from the monitored item can wait to be published at any given time. This setting depends greatly on the publishing interval of a server. If the server is slow to publish notifications to the client, a longer queue might be necessary to ensure that relevant information is not lost.

2.3 Alarms and Conditions

Alarms and Conditions (A&C) are ways that parts of an address space communicate their current state and possible risk or error states to clients and the rest of the system [5]. The OPC UA specification defines a condition as a sort of permanent event that always has a defined state, rather than occurring once and being disposed of after acknowledgment. Conditions represent the state of a system or a subset of a system. These are typically physical and practical, such as informing of a need for maintenance or other user action, or a value exceeding a defined limit. Thus, conditions are very important for communicating the states of the system to the user. A condition has two base states: "enabled" and "disabled". "Enabled" can be split into more specific sub-states, such as "value approaching dangerous level" and "value reached dangerous level" for a condition tracking a sensor value. When the condition is disabled, no event notifications will be generated for it. To be informed of changes in condition states, a client must subscribe to the condition. As seen in figure 3, conditions are attached to nodes with the HasCondition reference.

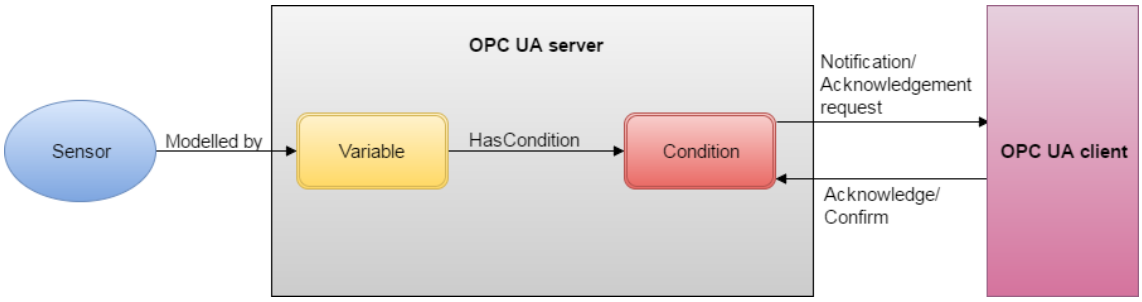

Figure 3: A typical use of a condition. A real sensor is modelled in the address space of the server, and a condition is created to monitor it and report to the client.

All conditions have a ConditionType, a ConditionClass and a ConditionSource [5]. Existing subtypes for conditions includes AcknowledgeableConditions, which have to be acknowledged by the user when their state becomes enabled, and dialogs which ask the user for input on behalf of the server. ConditionClasses classify which purpose the condition is intended for, e.g. ProcessConditionClassType for representing the state of a process and MaintenanceConditionClassType for representing a maintenance-related state. The ConditionSource determines what element, typically a variable or an object, the condition is based on or related to. Conditions have a severity, which tells how important the condition is for the overall performance and safety of the system. For instance, a condition that indicates whether temperature has reached a dangerous level should have a high severity.

Alarms are a specific kind of AcknowledgeableCondition that have additional states [5]. These states are active, shelved and suppressed. If the alarm is active, what the condition represents is currently happening, e.g. a value is beyond a safe limit. Active can be split further into sub-states. Shelved and suppressed both prevent the alarm from sending acknowledgment requests to the user. The difference

between the two is that suppressed is set by the server internally, whereas shelved is set by the user themselves. Their purpose is to prevent the user from being flooded with acknowledgment requests when the system encounters a problem and allows them to focus on solving the issue.

2.4 Historical Access

The part of the OPC UA standard that deals with past values is Historical Access (HA) [6]. HA is a feature that logs changes made in an OPC UA server address space. The historical data in HA can be stored in temporary memory buffer, or in a more permanent location such as a database. Different implementations store data at different rates and in different amounts, and typically vary depending on the context. Some values need more detailed archival than others. For instance, liquid level in an important tank might be recorded several times in a minute, but some values might be recorded only once per day.

An important aspect of historizing is the timestamp [6]. In a node it tells the time when its value was changed and for an event it tells when the event was generated. Variables have two timestamps: The source timestamp and the server timestamp. The idea is that the source timestamp comes from whichever device first generated the value first and will not be changed by the server, and the server timestamp is determined on the server itself [19].

A data node that the server can collect history for is called a HistoricalDataNode [6]. They are always a part of other nodes, typically as a property or a variable. A HistoricalDataNode has its Historizing attribute defined and can refer to a HistoricalDataConfigurationType object, which is explained in chapter 2.4.2. These can be seen in figure 4. The AccessLevel attribute of a node determines whether the history can be read or changed. The UserAccessLevel attribute determines the same for the currently connected user. The historizing attribute determines whether the history storing is currently enabled.

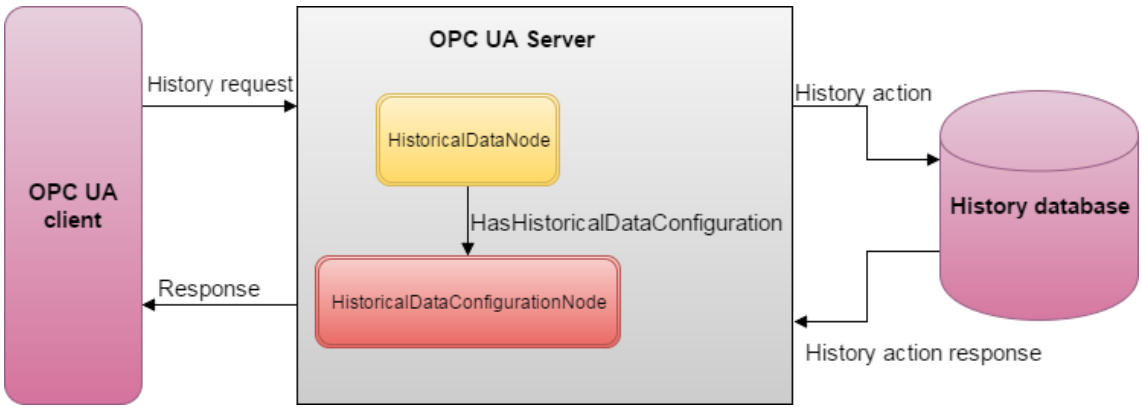

Figure 4: How an OPC UA server uses Historical Access. History values are read from and written to storage, typically as requested by a client.

In addition to data nodes, HA also allows the recording of events [6]. When a historizable event is generated, it can be stored with a HistorizedEventNode, which the server identifies by checking their EventNotifier attribute. As they are a subset of events, conditions can also be historized.

2.4.1 HA Services

As shown in figure 4, the client can make history service requests to the server. These services include reading, updating or deleting historical data.

Historized data is accessed and updated through the HistoryRead and HistoryUpdate services, respectively [20]. The same HistoryRead service is used for different purposes by changing the historyReadDetails parameter. By changing the parameter, HistoryRead can be used to either read recorded events from the database, raw (unmodified) history values, modified history values, aggregate values or interpolated values. Structured history data, such as annotations, are structures that in addition to a data value also contain a unique identifier for the data value. Some systems and cases allow a data node to have more than one value with the same timestamp, in which case the unique identifier is used to differentiate them.

If the number of historical values or events matching a HistoryRead request exceeds the maximum number defined in the request, the response to the request will return a continuation point for that can be used as the starting point for a further request [20]. A continuation point is also returned if the request handling has to be stopped prematurely due to taking too much time, buffer space or processing power.

HistoryUpdate is used to insert, modify, replace and delete stored values or events [20]. Similarly to HistoryRead, the same service is used for different purposes by changing the historyUpdateDetails parameter.

2.4.2 HistoricalDataConfigurationNode

A HistoricalDataConfigurationNode is a node that contains information on how history is stored for a particular node [6] and can be used by both the server internally as well as to inform the human user. It is linked to its node with the HasHistoricalConfiguration reference, as shown in figure 4,.

The structure of a HistoricalDataConfigurationNode is presented in figure 5. The node has one mandatory variable, Stepped, a boolean which defines whether the unknown value between two known data points should be estimated and graphically presented with a sloped interpolation (smooth lines between data points) or a stepped interpolation (flat horizontal lines between data points) [6].

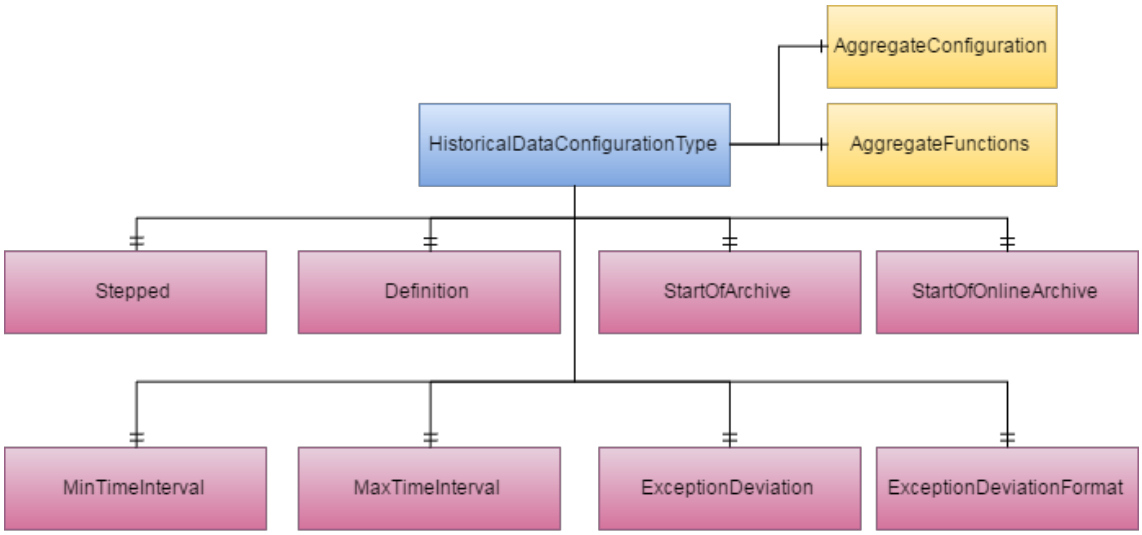

Figure 5: The structure of the HistoricalDataConfigurationType.

Its other, optional, variables are Definition, MaxTimeInterval, MinTimeInterval, ExceptionDeviation, ExceptionDeviationFormat, StartOfArchive and StartOfOnlineArchive [6]. Definition is a human-readable string that defines how the value of the associated node is calculated, typically in a form of an equation. MaxTimeInterval and MinTimeInterval are Duration variables that define the minimum and maximum times between two data points in value history, regardless of whether the value changes or not. This means that if the MaxTimeInterval is set, the server will store a value into history even if it did not change, and if MinTimeInterval is set, the server will not store a value until at least MinTimeInterval has passed since the storage of the last node. Similarly, the ExceptionDeviation variable defines the minimum amount that the value of the associated node has to change before it is recorded and ExceptionDeviationFormat specifies whether the ExceptionDeviation is absolute, a percentage of the previous value, a percentage of the value ranges defined by the instrument or the operation type, or unknown. StartOfArchive determines the earliest DateTime that the archive has values for either online or offline, and StartOfOnlineArchive determines the same specifically for the online archive.

3 Aggregating OPC UA server

Being able to access several different automation servers through a single software allows managing the data of a large number of devices and locations at once, which has led to the development of several technologies [1]. In a comparison between technologies allowing this kind of automation data aggregation [1], OPC UA was found to have several advantages, such as its built-in support for historical access, alarms and different communication protocols, such as a customized form of TCP and HTTP.

To achieve such vertical integration, OPC UA servers can be connected and organized in a layered architecture, as described in the OPC UA specification [2]. Higher-layer servers collect and organize data from the lower layer servers, and the client can access and interact with the total system through the highest layer. This kind of architecture is called the aggregating server architecture [4]. Aggregating servers can themselves be aggregated, so the structure can have more than two layers.

Figure 6 depicts an aggregating OPC UA server. It interfaces with source servers through their clients, and external systems access the data of the source servers through the client of the aggregating server.

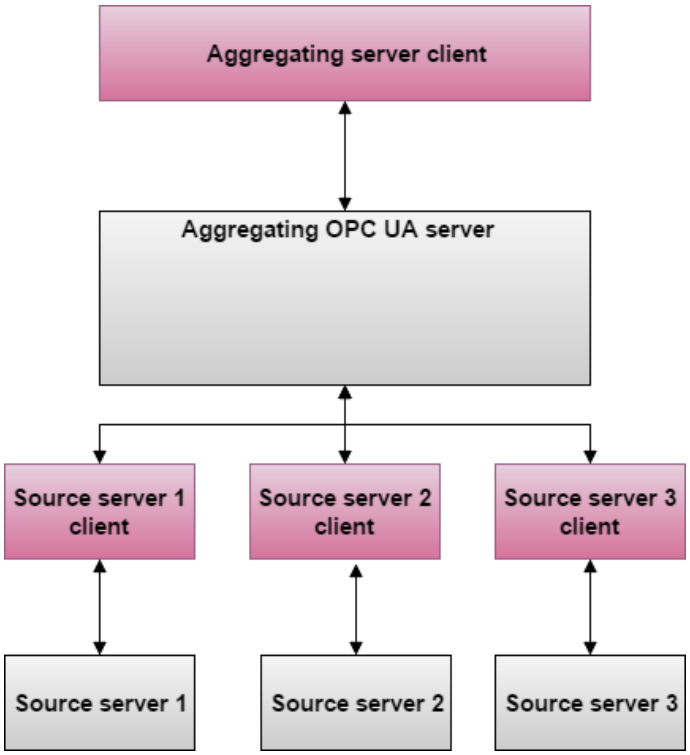

Figure 6: An example of an aggregating OPC UA server that aggregates three source servers.

3.1 OPC UA address space transformation

The OPC UA standard defines an information model as an organizational framework that defines, characterizes and relates information resources of a given system or set of systems" [21]. An information model is one representation of the structure of the information in a system or a set of systems [22]. However, it is not the only possible representation of that data. Information model transformation is taking the data represented by one information model and restructuring it to match another information model. This means that while the first model and second model are based on the same data but are structured differently. It could be as simple as changing the relations or hierarchies between objects, but sometimes the new information model demands different types of objects.

Consider an environment like a factory, with hundreds to thousands of devices, each consisting of multiple parts, and a server collecting data from all of them. If all the data is left as it is, the resulting server is very difficult to manage. What is needed is an information model transformation. In the case of OPC UA, an address space transformation can be used to transform the address spaces of the source servers into the aggregating server address space [4]. Transforming one OPC UA address space into another can help show and gather exactly the information that the client accessing the server needs.

When data is collected from a lower-layer system to a higher-layer one, there is a chance to change the structure of the data in some fashion. For example, If a client only needs data from some of the sensors on the lower-layer servers, the higher-layer servers should only take the sensor data from the lower layers and organize it in some readable fashion. From the perspective of the client accessing the highest layer, there is only one server.

An address space transformation in aggregating servers is not necessarily one-to-one. What this means is that what would be portrayed by several nodes in one model might be portrayed by just one in another. For instance, some group of nodes on lower-level servers might only be represented by the calculated average of their data values on the aggregating server [10]. Some transformations could also require more complicated m-to-n transformations, where values and references from m source nodes are used to create n nodes in the target address space. Thus, references are also valuable information for the transformation process.

The structure of an OPC UA address space can contain cycles, as not all References are hierarchical [3]. Depending on the algorithm used for browsing the address space, this can lead to revisiting Nodes or even infinite loops. As the NodeId of each node is unique, storing visited NodeIds in e.g. a list and comparing the NodeId of the current node to them is a possible if potentially time-consuming way to avoid loops.

An OPC UA address space transformation will have to take into account that things are represented differently on different servers. Type mapping rules translate one type to its equivalent on another server, or identify semantically equivalent types on different servers [4]. As Types can inherit from other Types, the transformation process needs to make decisions of which rules are applied to which types. Instance mapping rules deal with the handling of objects on servers. One example of this is if

several source servers each have an identical object instance, of which the aggregating server only needs one [4].

As a simple example, consider an aggregating server that aggregates two source servers, A and B, representing different parts of the system but in the same environment. As seen in Figure 7, The valves of servers A and B are slightly different and are thus modelled as ValveAType and ValveBType, but for the aggregating server and the end user the type differences do not matter and both are mapped as simply ValveType. Also, both A and B have included the same room temperature sensor, but the aggregating server does not need the same information twice and only maps it once.

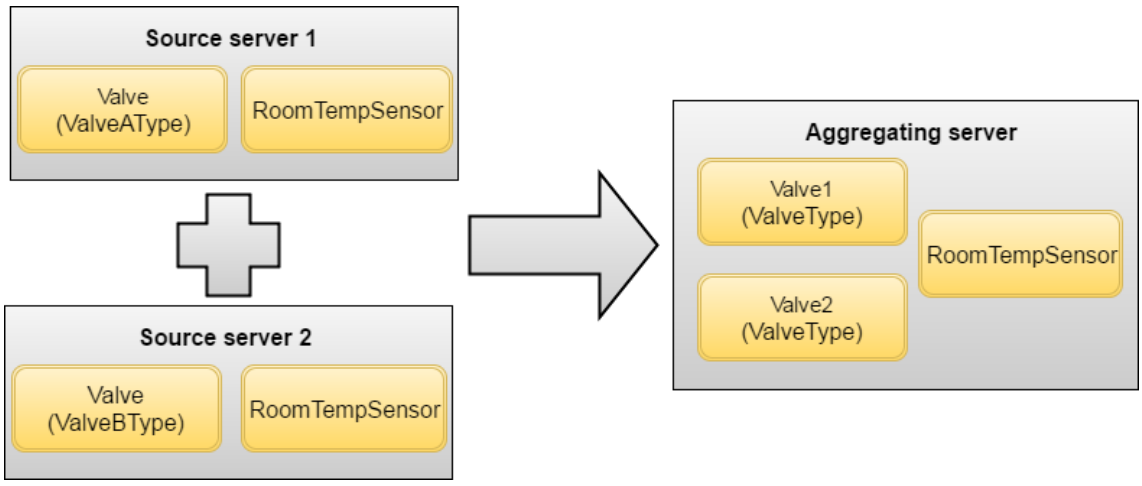

Figure 7: An example of address space transformation dealing with different typing and redundant nodes on source servers.

3.2 Related research and implementations

3.2.1 OPC UA Historian

The OPC UA Historian [10] is an aggregating OPC UA server developed by Prosys. It is used as to collect and access the current and historical data of other OPC UA servers that do not implement Historical Access themselves. The historical data is stored in an SQL database. The OPC UA Historian allows the user to select which nodes are to be historized, and also allows later removal of nodes. However, it currently does not allow fully changing how the source server address space is transformed and mapped to the aggregating server.

3.2.2 The aggregating server of Ingolstadt university

Researchers in Ingolstadt university in Germany[4] proposed a manner of address space transformation where mapping rules are integrated into OPC UA information model extensions. In this transformation model, source servers are responsible for

mapping which source server node types match which aggregating server source types, and how aggregating server NodeIds are mapped to source server NodeIds.

An aggregating OPC UA server using this model was made. The server could relay some service calls, such as reading and writing, from the aggregating server to the source servers using its mapping [4].

3.3 Aalto University aggregating OPC UA server

The Ingolstadt university aggregating OPC UA server model served as a basis for an aggregating OPC UA server developed in Aalto University in Finland, using the Prosys Java SDK for OPC UA [23].

For each server this server aggregates, it creates a NodeManager, which is in charge of relaying service calls from aggregating server nodes to the source server nodes they correspond to. NodeManagers do this by keeping a record of which source server they are associated with, as well as a hashmap-based mapping of aggregating server NodeIds to source server NodeIds. For example, as seen in Figure 8, a Read call at a node on the aggregating server would be caught by an IoManagerListener, relayed by a NodeManager as a Read call to the client of the aggregated server containing the original node, and the result response of that server would be relayed back to the original client. None of this is visible to the external system asking data from the client, all it can tell is that it made a read call and received a result.

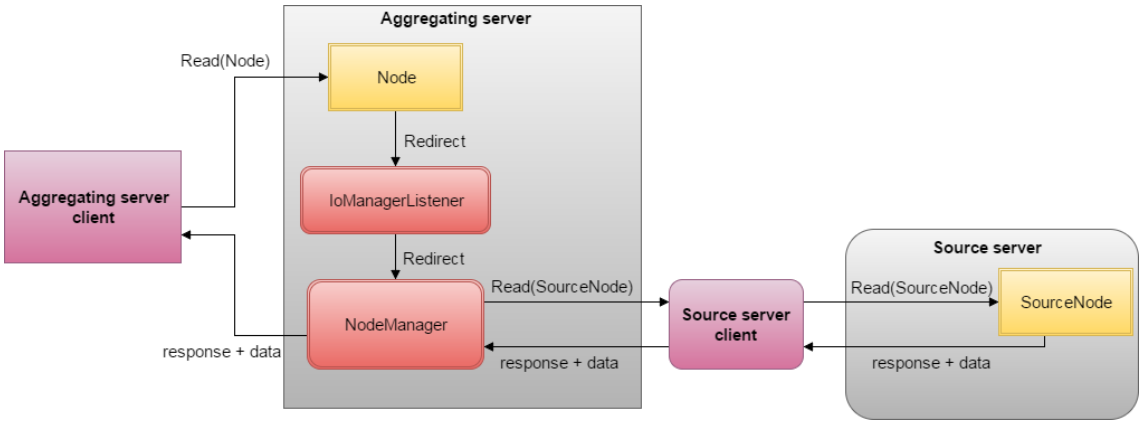

Figure 8: How the Aalto University aggregating OPC UA server relays a read call. The NodeManager is used to find the original source node and its server.

In addition to the aforementioned NodeManager and IoManagerListener, the aggregating server developed at Aalto University relies on the NodeManagerListener of the server and the MonitoredDataItemListener of its internal client to use subscriptions to notice changes made to values on the aggregated server and update the values on the aggregating server accordingly. This is discussed in more detail in the master's thesis of the developer [23].

The transformation algorithm used by the server was influenced by Extensible Stylesheet Language Transformations (XSLT), the Backus-Naur Form (BNF) and

regular expressions. The algorithm is capable of transforming one address space into another, but is limited to regular Data Access, and one-to-one transformations [8]. It is thus able to alter the reference structure of a model and leave elements out, but it is currently not capable of combining nodes.

3.3.1 Transformation algorithm

The address space transformation method used in this work is based directly on the previous version of the aggregating server [8], but expanded to deal with e.g. history handling. The configuration of this aggregating server is based on going through the source server(s) and using a pattern-recognition algorithm based on regular expressions. This finds node-reference structures and maps them to different structures on the aggregating server.

Information model transformation rules have two parts: The left-hand side (LHS), representing the original source model, and the right-hand side (RHS), representing the target model. In the scope of this thesis, it is enough to understand that a transformation algorithm takes the source model as an input and rearranges it into the target model based on the difference between LHS and RHS.

The model transformation algorithm uses a regular expression-like combination of string patterns and variables on both LHS and RHS and the algorithm uses those for mapping the transformation [8]. The rules are contained in a file separate from the rest of the source code, allowing the rules to be changed in a modular fashion without affecting the software itself. A rule manager is used to parse the rule strings and instruct the software on how to enact them. Consider the following rule as an example:

- LHS: [BoilerType]#1/FT1001#2
- RHS: #1/[PipeType]Pipe1001/[FTType]FT1001/#2(BrowseName=DataItem, DisplayName=DataItem)

The algorithm is browsing the source address space, using Organizes and HasComponent references to find new nodes, chosen because of how commonly they are used and because when properly used, they should not cause loops while browsing the address space. On the LHS, the algorithm recognizes a BoilerType node with a child node named FT101 and assigns them as variables #1 and #2, respectively. On the RHS, the algorithm creates a slightly different structure where it copies and rearranges the data, creating a PipeType and FTType node in between the BoilerType and FT1001 nodes. The transformation algorithm and its syntax are explained in more detail elsewhere [8] .

4 Alarms and conditions

4.1 Requirements

As the aggregating server serves as an interface to the servers it aggregates, it should be able to pass on information about the conditions and alarms of the system to its clients. This is implemented by having conditions on the aggregating server that are triggered by changes on the source servers. An aggregating server can even be made primarily for condition notifications, e.g. notifying a factory monitor of situations requiring manual intervention. Higher-layer conditions can possibly use lower-layer conditions as inputs. This would allow further customization of the information presented to the user.

The aggregating server needs to fulfill the requirements set by the OPC UA Alarms And Conditions (A&C) specification [5], such as A&C-related methods.

4.1.1 Cases

There are three possible scenarios for condition generation on the aggregating server. Each has different requirements, advantages and disadvantages.

The first case is that the source server has a condition which is transformed and mapped to the aggregating server as a proxy condition. The new condition is then connected to the source server condition via subscription. Figure 9 illustrates this case. An example of this scenario would be an aggregating server that aggregates all the different conditions from across the system for monitoring purposes. Regardless of the source server condition they are based on, the function of the aggregating server conditions can be the same, as they simply need to be triggered by the original condition, thus only serving as proxies for the original conditions. As such, this is a simple and light solution, as all conditions can be treated in the same way.

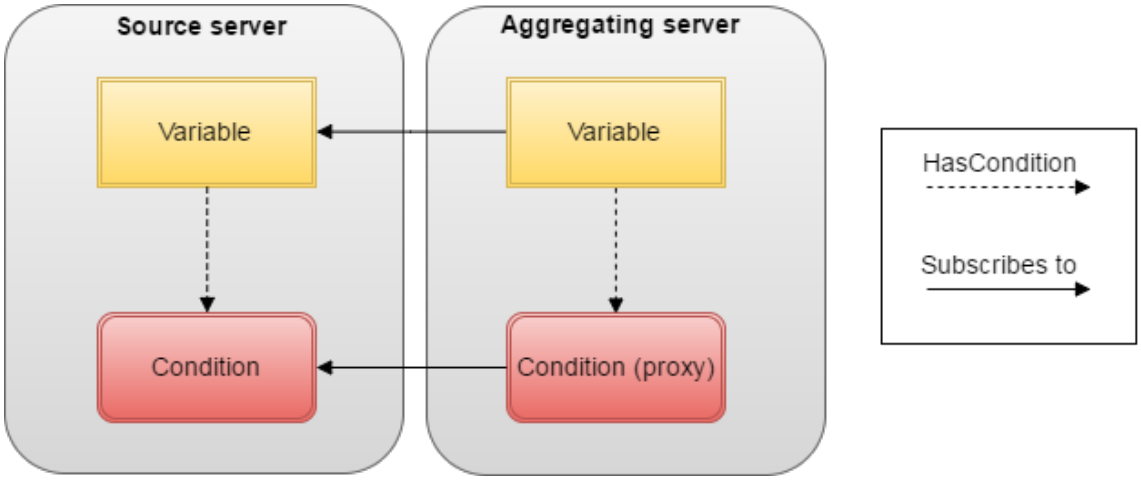

Figure 9: An aggregating server with proxy conditions.

Proxy conditions have no internal logic of their own, merely tracking the state of the source server condition they subscribe to. The alternative of copying the functionality of a source server condition through its client is not feasible, as it would require replicating another piece of software during its runtime. Even if there was a way to do that reliably, there is no guarantee that the source server condition is not itself reliant on some external software such as a database or a third server. Because of these factors, this possibility was not explored in this work.

Dialogs are conditions that prompt the user for input. As the dialog prompts are sent to the user through the server client, relaying prompts from the source server to the aggregating server would require a way to detect when they occur and then send the prompt to the aggregating server client. The user would then respond to the prompt and the response would be relayed back to the source server client.

The second case is that the aggregating server generates a local condition based on source server values, but no equivalent condition exists on a source server. In the proxy condition case, it was enough for the transformation algorithm to notice a condition-type node on the source server. In the case of local condition-handling, the transformation algorithm instead needs to search the server for nodes and variables that require conditions to track them. Upon finding such a variable, the transformation algorithm creates an equivalent aggregating server variable and a condition and connects the two. The result is shown in figure 10.

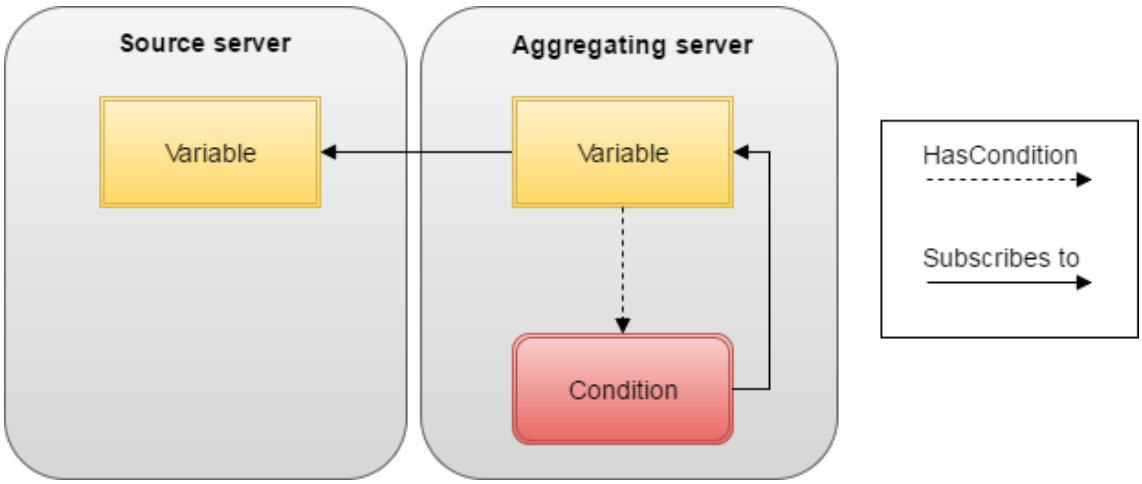

Figure 10: An aggregating server with its own local conditions. Local conditions do not interact with source servers directly, only indirectly through local variables.

An example of this kind of use would be an aggregating server that aggregates a source server representing a device, designed by a third party. On that source server is a variable, e.g. a flow measurement. For the source server, the value of the variable is not critical, but to the larger system it might be. Other devices in the system might be incapable of handling very fast flows. So, a condition to track the flow is necessary, and it is created on the aggregating server and connected to a variable on the source server.

One advantage of local conditions is that A&C-methods can be used exactly as in a single-server solution, since the required condition nodes and logic are are all on the aggregating server. This also centralizes the A&C-logic to one server, meaning that troubleshooting and modifying the A&C can be done entirely on the aggregating server.

The third case is that the aggregating server has both relaying proxy conditions and local conditions. This case is shown in figure 11 This requires the aggregating server to keep track of which condition is local and which is a proxy, so that method calls can be handled correctly. This case would be used when source servers have A&C, but it is insufficient for the larger system and requires local conditions to supplement the proxy conditions.

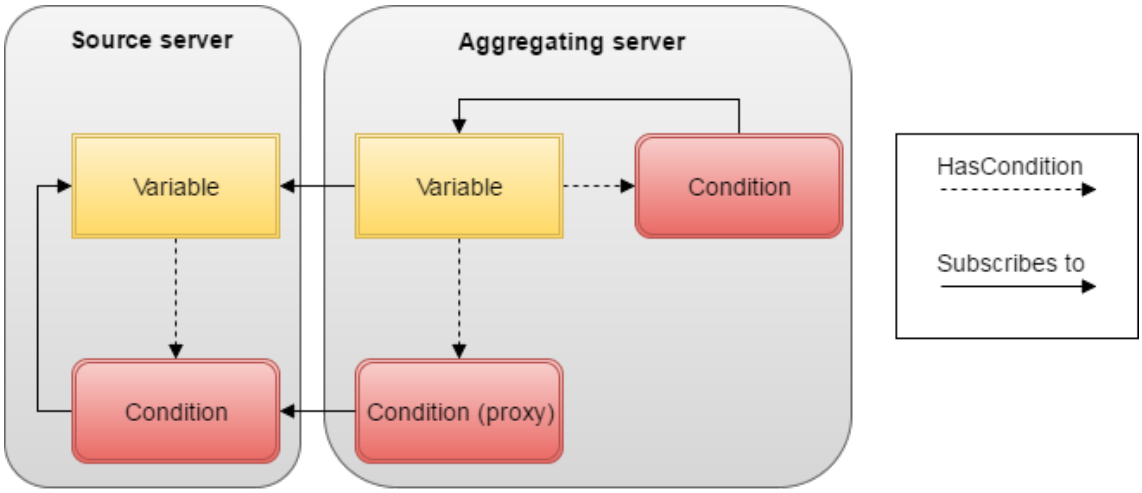

Figure 11: An aggregating server with a combination of relayed and local conditions, a hybrid of the previous two designs.

4.2 Design

The A&C design in this work uses proxy conditions as described in section 4.1.1. This means that during the configuration of the aggregating server, the transformation algorithm will try to find conditions on the source server and then create aggregating server equivalents and connect the two through subscription.

4.2.1 Transformation algorithm

Variables are connected to conditions associated with them through HasCondition references. When browsing through a source server, the transformation algorithm checks the references of each Variable it visits for HasCondition references, thus finding the conditions of the source servers. For each of these alarms, a function called CreateCondition is called.

CreateCondition creates a condition node with the same parameters as the source condition, such as messages, severity and triggering limits. The string value ActiveState of the new aggregating server condition is set to Subscribe to the ActiveState of the source condition. This means the state of the aggregating server condition will follow that of its source server equivalent.

4.2.2 Relayed alarms and conditions

The designs of how A&C-related methods for proxy conditions are described in table 1. As long as a mapping exists between the aggregating server condition and its source server equivalent, method calls such as Acknowledge, Confirm, Disable and Refresh can be relayed to the source condition. The source condition can then change its status, leading the the aggregating condition following suit. The Shelve and AddComment methods deal with what the user perceives regarding the condition and should be focused on higher-layer servers rather than lower-layer servers that the user does not directly interact with.

Table 1: A table about the different methods and features related to A&C and how they work with proxy condtions.

	Description	**Design for proxy conditions**
Acknowledge	When an condition is Acknowledged, it signals that the operator has noticed it and will try to address it.	Acknowledge on aggregating server. Use mapping to send acknowledge call to source server.
Confirm	When an alarm is Confirmed, it signals that the operator has dealt with the source of the issue.	Confirm on aggregating server. Use mapping to send confirm call to source server.
Disable	When an alarm is disabled, it cannot be triggered by anything.	Disable on aggregating server. Use mapping to send disable call to source server.
Refresh	Refreshing calls for every Condition with Retain set as True to send their current statuses as events to their subscribers.	Refresh on aggregating server. Use mapping to send refresh call to source server.
Shelve	When an alarm is shelved, it is temporarily prevented from being shown to the operator	Shelve on the aggregating server.
AddComment	Attaches a text comment to a condition's state. The comment might be e.g. details about what has or needs to be done to change the state.	Add the comment on the aggregating server.
Dialogs	Dialogs are Conditions used for asking user input.	Notice prompt sent to source server client. Request user input on the aggregating server. Use mapping to send response to the source server.

5 Historical access

5.1 Requirements

An aggregating server that implements Historical Access (HA) should follow the OPC UA Historical Access specification. As such, it should implement HA services and methods as instructed in the specification [6]. Regardless of how the HA is implemented, the server should be able to write and read historical data, recording changes to the values. Chapter 5.1.1 examines these services in more detail, as well as how their implementation depends on the type of history storage used.

The history manager referred to in this chapter is not a part of the OPC UA specification itself. However, each aggregating server that has HA-features should have some part of its software dedicated to managing HA requests. That part is called the history manager in both the C++ by Unified Automation [15] and Java software development kits for OPC UA by Prosys [9] and was adopted for this work as well, though this work is not exclusively about those implementations of OPC UA.

5.1.1 Cases

There are three basic forms of HA on aggregating servers, which place different requirements on the system. The historizing process and historical data storage of HA can be performed on the on the aggregated source server, the aggregating server or on both. Each of the three cases places different requirements for the implementation.

The first case is relayed history storage on source servers. It is shown in Figure 12. Rather than having its own history database, the aggregating server outsources its history storage to the source servers. While the image has the source server with its own history database, the server itself might similarly outsource its history storage to another server. The details of how the source server handles its own history management are not relevant to the aggregating server. When the aggregating server client makes a history request such as HistoryRead for a specific node, the request is relayed to the source server the node was aggregated from. As this solution has no requirements on a history database, it is the simplest and lightest option.

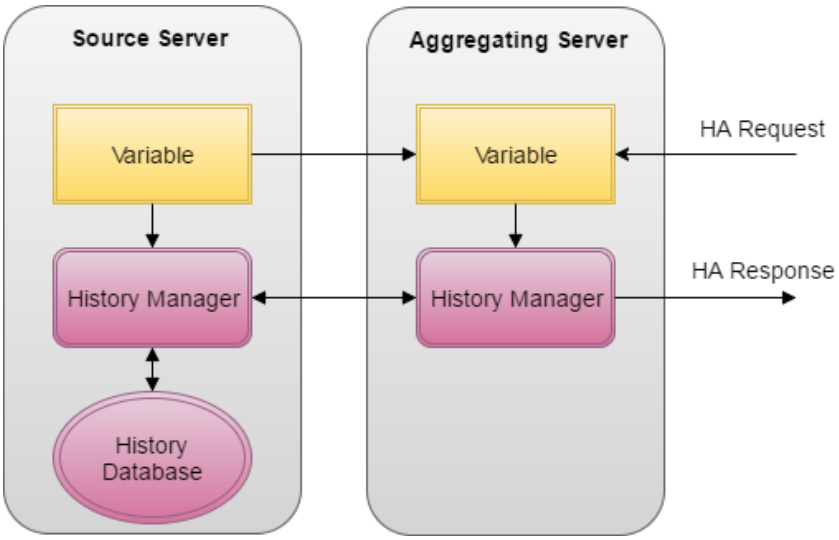

Figure 12: History data storage on the source server.

Relayed history operations have difficulties that local history operations do not. The core issues are the handling of interpolation and aggregate values, as those are server-specific [6]. Relaying ReadProcessedDetails and ReadAtTimeDetails calls can lead to severe inconsistencies as each source server calculates interpolated and aggregate values in their own way. Performing the interpolation on the aggregating server is difficult, as it has no direct access to the history databases of the source servers, which means searching for the closest earlier and later values is not trivial. Another potential problem with ReadProcessedDetails is that it can be used to request historical aggregate values based on nodes originating from different source servers. Thus, ReadProcessedDetails cannot be implemented with a simple relayed request. A possible solution is retrieving the required raw data from the different source servers with relayed ReadRawModifiedDetails calls, calculating the requested aggregate values on the aggregating server and returning them.

The second case is local history storage on the aggregating server, as depicted in Figure 13. If HA is only on the aggregating server, nodes and events on the aggregating server based on lower-layer data save their values into some sort of historical database. This type of history storage has been examined in some detail. Examples of these kinds of solutions include the OPC UA Historian [10] and a the OPC UA History Gateway [24]. In these implementations the aggregated server does all the historizing in the system, keeping an extensive database of past values. This is a way to centralize the history management and allow easy access to data for analysis. If the values that the are meant to be historized are updated on the aggregating server in a timely fashion, this approach should not differ from implementing HA on a single non-aggregating server.

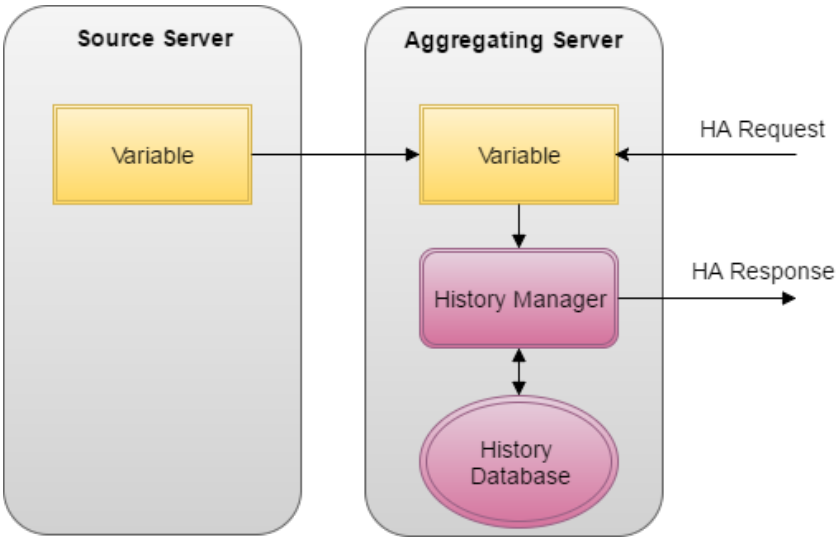

Figure 13: History data storage on the aggregating server.

The third case is a hybrid of the previously explained local and relayed history storages, as shown in Figure 14. If historizing is done both on aggregating and aggregated servers, each server might implement it differently. For example, different layers might store values at different rates or have a different buffer size for historical values. An example of this would be a case where the aggregated server stores detailed history data from a shorter period, such as the last 24 hours, but the aggregating server might have data from a longer period but only a few values per day. Another case of differing HA implementations on the server would be simply storing history values for data that is not historized on the aggregated servers. For instance, if the aggregated servers are ready-made commercial solutions, some might have their own history databases while others might not, and the aggregating server can provide history storage for the latter.

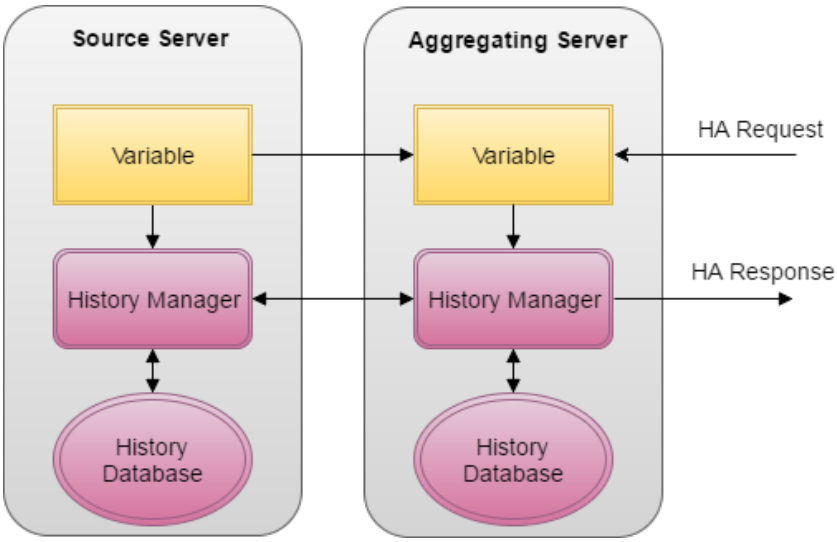

Figure 14: Hybrid history data storage, with history data on both the aggregating server and the source server.

History operations in hybrid implementations will require accessing history data on both the aggregating server history database as well as on the source servers. HistoryRead requests require the combination of data from both the aggregating server and the source servers. HistoryDelete operations should be done on both aggregating and source servers to ensure deletion. In the case of modifying or updating a historized value, the aggregating server history database should be checked for a value with the given timestamp. If it exists, the operation is done to that value. Otherwise, the HistoryUpdate or HistoryModify request is relayed onto the source server.

5.2 Design

5.2.1 Transformation algorithm

History handling options were added to the transformation rule system of the previous aggregating server[8], in addition to the previous parameters LHS (Left-Hand Side) and RHS (Right-Hand Side). The history handling parameters are optional. If the rule has missing history handling instructions, the server default settings are used.

The first history handling parameter is the history storage location, the most important one. There are three different string options: "local", "relay" or "none". The design for the hybrid case is not explored in this section.

In all cases, the history storage type is written into the HistoryConfiguration node of the created node. In addition, if the parameter is "local", the HistoryManagerListener of the aggregating server is called to create a local history for the node. If the user defines the history storage as "local", they can also add the parameters minTime and maxTime, which are respectively the minimum and maximum times that can pass before the server records the value of the node into history. These

are written into the HistoryConfiguration of the node as well. The minTime- and maxTime-parameters allow different nodes to have different history storage intervals.

As in the previous version of the rule system, each aggregating server node only has a maximum of one corresponding node on a source server. This means that that processed values such as averages or sums of multiple source server node values are currently not supported. This also simplifies the history handling system, as any history request on the aggregating server can only result in one history request on a source server. A many-to-one transformation would require significantly more mapping information and a more elaborate rule system.

5.2.2 History handling types

Figure 15 depicts how the aggregating server determines how it chooses how to handle history calls for a node. During runtime, the server recognizes whether nodes have their history access enabled by checking their AccessLevel attribute [6]. The AccessLevel has eight bits that each describe whether a procedure is allowed for the node. If the bit for HistoryRead is marked as 1, HistoryRead is allowed for the node, and similarly for HistoryWrite.

When a call of that type is made for the node, the history manager checks the HistoricalDataConfiguration of the node: If the history storage for the node is marked as "local", the history manager will perform the action in the history database of the aggregating server. If the history storage of the node is marked as "relay", the HistoryManagerListener must then determine which node on which source server corresponds to the aggregating server node.

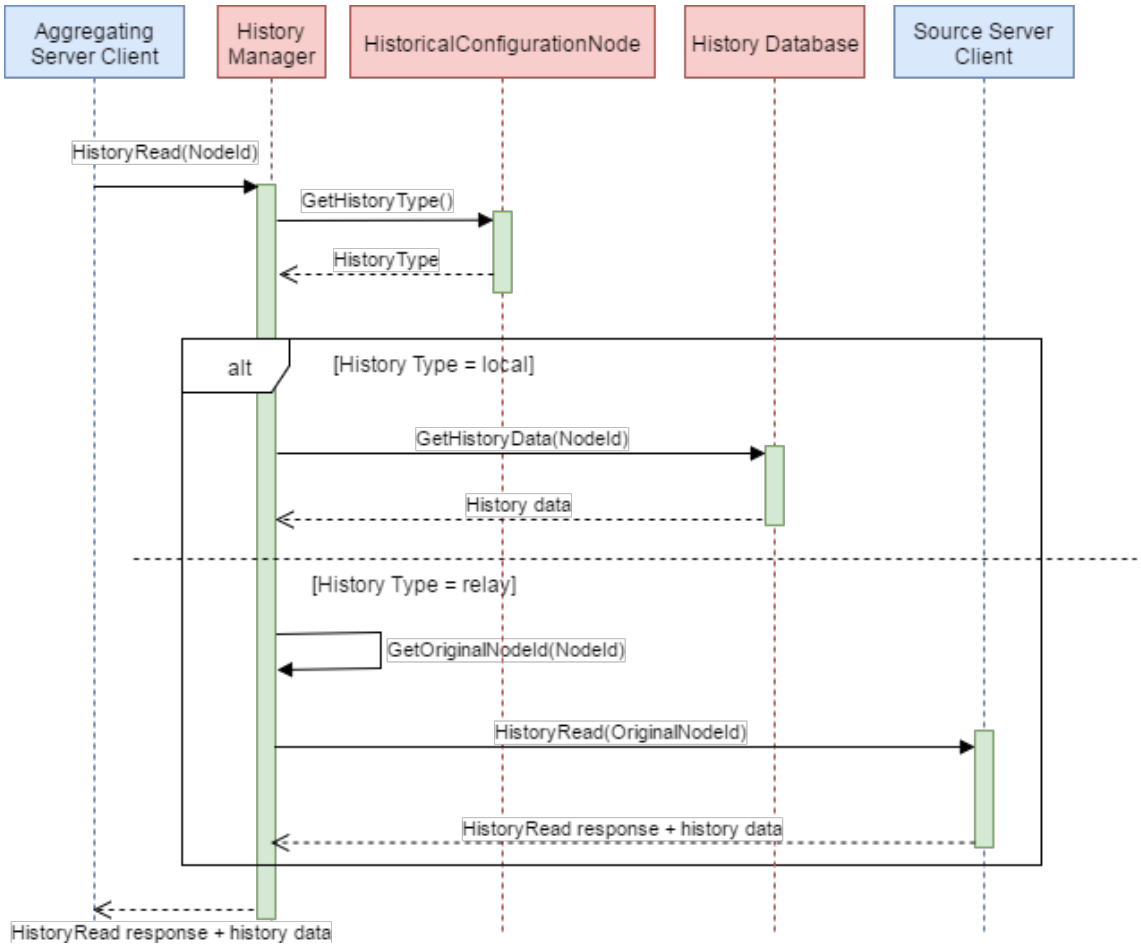

Figure 15: UML sequence diagram of how the aggregating server handles a HistoryRead call. The process depends on whether the history storage of the node is local or relayed.

In the relayed case, the getOriginalNodeId-function consults the mapping of the aggregating server to find the source server the node was transformed from, as well as the original NodeId of the source node. A new history service call with the same parameters is then made to the source server, using the found original NodeId.

Historical Access services are handled differently in each of the three cases. In the case where history data is stored on the source servers, most services are done through relaying the requests. ReadProcessedDetails and ReadAtTimeDetails require the aggregating server to request the needed raw data from the source servers and then perform the necessary calculation or interpolation locally. For ReadAtTimeDetails, if no raw data is found for the given timestamp, new requests for raw data are made with an earlier starting time and later ending time until one earlier and one later stored value are found for the interpolation. In the case where history is stored locally on the aggregating server, all services are handled as in a single-server case.

6 Performance evaluation

6.1 Evaluation system

The server setup used for all tests uses one or more source servers and one aggregating server. The source servers are based on the concept of an industrial setting with one or more boiler-devices. These Boiler servers are based on the ones used in previous work on this aggregating server [8], with some differences. As the new Boiler servers are used for testing HA and A&C, they now have a basic alarm for each boiler device as well as a working historizing implementation. In addition, a basic method was added for testing method call relaying from aggregating server to source server. An example of the transformation of a one-boiler source server address space to the aggregating server address space is shown in figure 16. Each boiler-object in the source server address space is transformed into a smaller object with fewer child nodes in the aggregating server address space.

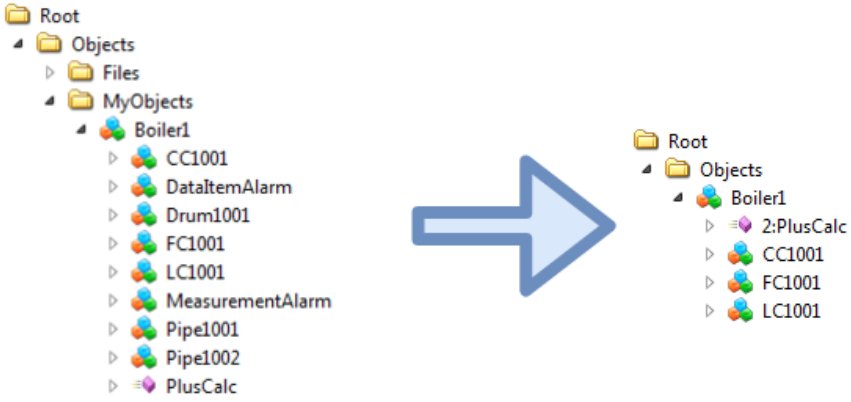

Figure 16: Comparison of the address spaces of the source server (left) and the aggregating server (right).

The task of the transformation algorithm in these tests is to transform multiple Boiler servers into one aggregating server containing the relevant data from all the boiler-devices on the source servers. During runtime, the transformed values on the aggregating server will continuously be updated through subscriptions as their source value equivalents change. The aggregating server interacts with the source servers as necessary during runtime, e.g. to read current or historical data.

All tests in this chapter were programmed and executed with the Java programming language and the Prosys OPC UA Java Software Development Kit. The software used for running the actual Java OPC UA servers was Eclipse Java Mars version 4.5.2.

The same desktop PC was used to run all the servers and the tests in this work. Its specifications were:

- Operating System: Windows 7 Home Premium, 64-bit
- Processor: AMD Phenom(tm) II X4 945 Processor, 3.00 GHz
- RAM: 4,00 GB

6.2 Configuration

Aggregating server configuration tests were run for a scenario with a single source server of varying size as well as one with multiple source servers. For testing aggregating multiple source servers, several identical instances of a source server were started. Each of these source servers has 8200 nodes. An aggregating server was started and the time it took to map and transform the source servers was measured. This process was repeated for 10 iterations for each number of source servers. The average transformation times are shown in figure 17. The transformation time rises linearly with the number of servers at roughly the same rate as in the single-server aggregation.

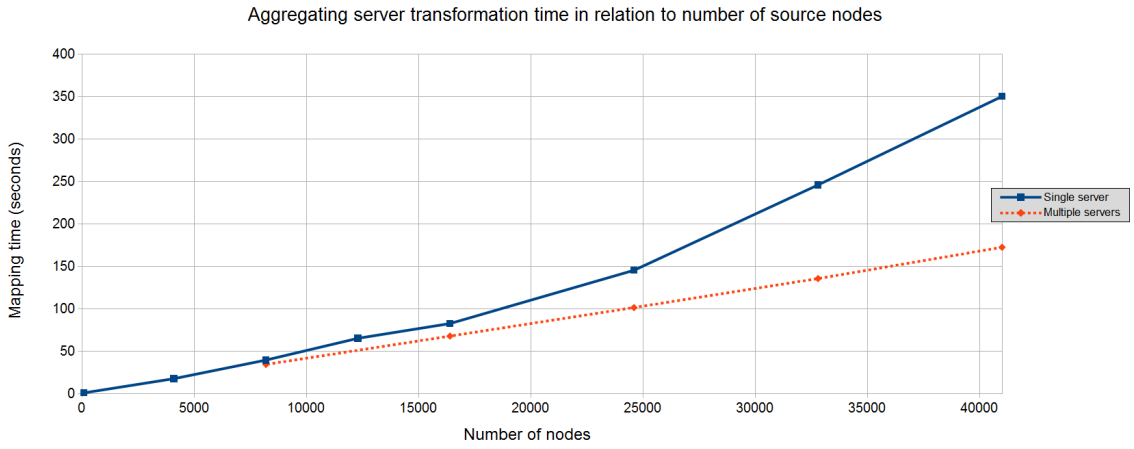

Figure 17: Aggregating server transformation times in relation to number of nodes on the source server.

Increasing the number of source servers seems to be less demanding for the transformation algorithm than increasing the size of a single source server. Increasing the number of source servers increases the transformation time linearly, but increasing the number of nodes on the source server in the single server case begins slowing down the transformation process at the 16400 node mark.

Attempts to test larger source servers failed due to hardware capacity-related errors halting the testing process.

6.3 Data access

Testing the Data Access performance was done by running Data Access service requests to a server multiple times and measuring the time taken for each request. The times taken were measured separately on both a source server and an aggregating server that had aggregated it. The size of the address space on the tested servers was varied to find how address space size affects data access efficiency.

Requests made to nodes on the aggregating server are relayed forward to equivalent nodes on the source server, if they exist. This means that the time taken by each request to a transformed node on an aggregated server also includes the time taken by the corresponding request that has to be made to the source server.

6.3.1 Read

The testing process of the Read service was done by measuring the time it took for the server to read 100 Double type variable values. This process was repeated 100 times. The resulting times were saved into a table file. To increase efficiency, the Read requests were sent to the server for batches of 100 nodes rather than sending a separate request for each node.

Figure 18 shows the average times taken relative to address space size. In the tests, the aggregating server was 25-30 ms slower at all address space sizes. On the source server, the time taken did not increase relative to the address space size. The aggregating server performance is more uneven, but still does not increase consistently with server size.

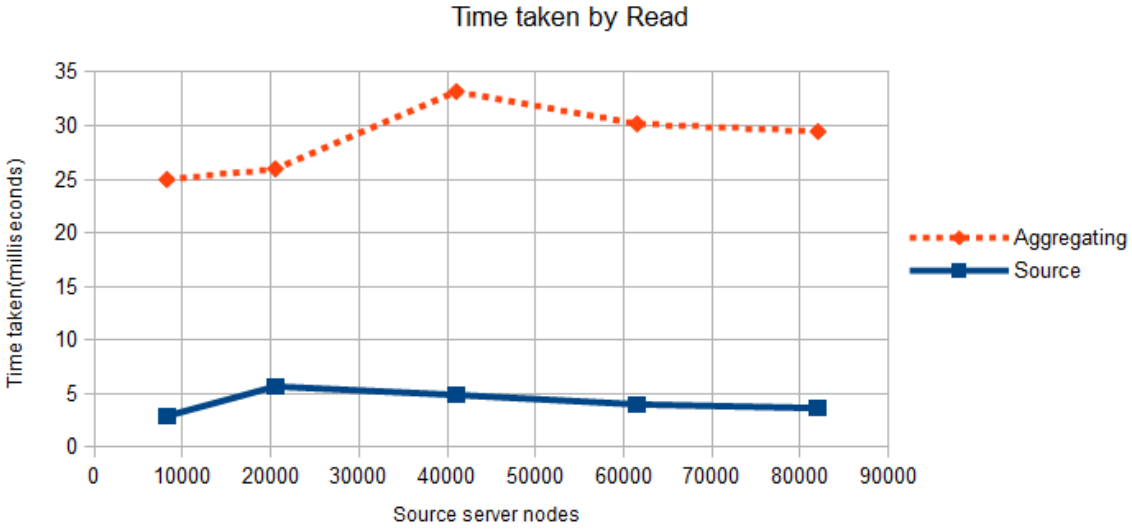

Figure 18: Average time taken by the read service on both source and aggregating servers, in relation to number of source server nodes.

6.3.2 Write

The Write service testing process was done by measuring the time it took for the server to write 100 Double type variable values. This process was repeated 100 times. The resulting times were saved into a table file. To increase efficiency, the Write requests were sent to the server for batches of 100 nodes rather than sending a separate request for each node.

As seen in figure 19, the aggregating server is again noticeably slower, with slight unevenness. In comparison, the times taken on the source server remain almost exactly the same. The aggregating server is shown to be around 25-40 ms slower than the source server with the tested address space sizes.

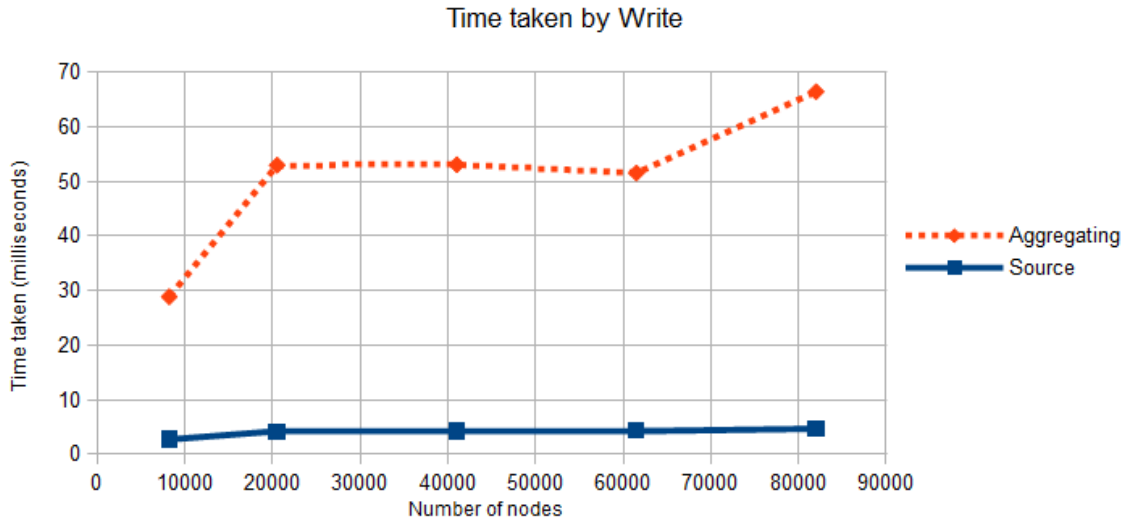

Figure 19: Average time taken by the write service on both source and aggregating servers, in relation to number of source server nodes.

6.4 Historical access

6.4.1 HistoryRead

To set up the HistoryRead tests, directly after the configuration of the source server, each of the 100 variable values to be tested were written over 10 times with a random Double value to create a history to read. The HistoryRead service testing process was done by measuring the time it took for the server to read the histories of 100 variable values. This HistoryRead was repeated 100 times. The resulting times were saved into a table file. Unfortunately, the Prosys OPC UA Java SDK did not allow making HistoryRead requests in batches, so the service was instead called separately for each tested variable.

In a setup with a single source server, the number of nodes on the source servers was varied to examine how the number of nodes in an address space affects the time required to read the history data. The testing was performed both directly on the source server and through relayed write requests on the aggregating server.

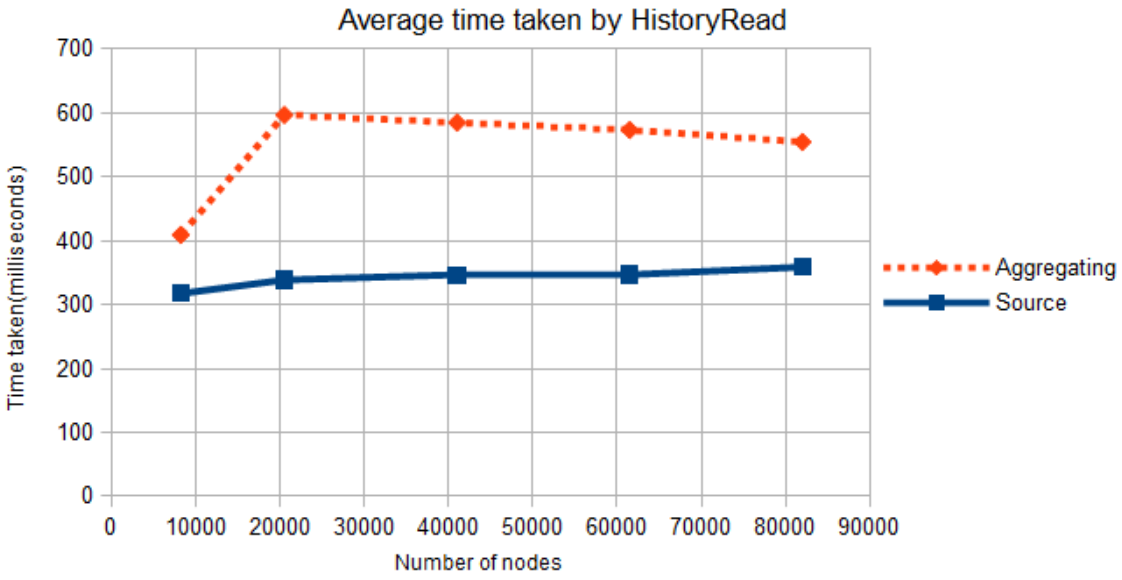

Figure 20: Time costs of reading history data on both source and aggregating servers, in relation to number of source server nodes.

7 Conclusions

This thesis set out to examine how aggregating OPC UA servers could incorporate Historical Access (HA) and Alarms And Conditions (A&C), as well as evaluating the performance of an aggregating server prototype. The thesis started with a literature study in chapters 2 and 3 about relevant OPC UA concepts as well as the concept of aggregating servers in general. After that, the requirements and designs for Alarms And Conditions (A&C) and Historical Access (HA) on aggregating OPC UA servers were studied and explained in chapters 4 and 5, respectively. Finally, the performance of a prototype aggregating OPC UA server was tested and evaluated.

Based on the design for proxy conditions, implementing A&C on an aggregating server should not be difficult. Thanks to the proxy condition subscribing to the state of the source condition, emulating the functionality of the original condition was not necessary for tracking the state. A&C methods, such as acknowledging, can be done by relaying the call to the original source condition. Unfortunately, complete testing of the A&C design was not possible due to unresolved errors when creating condition nodes on the aggregating server. However, basic testing was done using a node with the attributes and properties of a condition, but not the type. These tests proved that a node on the aggregating server could track the state of a condition on the source server.

The relayed historical access design detailed in this thesis works well for simple reading and updating of historical data. Most calls to nodes on the aggregating server can simply be relayed to the original source nodes. The ReadProcessedDetails and ReadAtTimeDetails services require special attention due to different servers handling aggregate values and interpolation differently. To ensure that they are calculated consistently, they need to be calculated on the aggregating server. This means the aggregating server needs to retrieve each raw value required by the calculations from the source servers, which takes much more time than a simple relayed service call or a local service call. Thus, an aggregating server using relayed history access is likely slower at those services than one using local history handling. The hybrid history storage case would be complicated to implement, with each service requiring the aggregating server to determine where the history data should be stored. As such, the hybrid case should probably not be considered unless the relayed or local history options by themselves are insufficient.

7.1 Result analysis

The aggregating server consistently takes multiple times longer than its source server to fulfill service requests during runtime. However, the performance of neither server seems to be affected by the size of their address spaces. This is useful to know, as thus the scaling issues of aggregating servers are concentrated in the configuration phase. Most servers do not need to be configured often, so even a large aggregating server will only rarely take up extra time. When fulfilling a service request, the aggregating server relays the request to a source server and waits for its response. Test times measured on the aggregating server therefore also include the time taken

by the source server, so the aggregating server is always slower than the source server.

The services tested were Read, Write and HistoryRead. The aggregating server was 7.01 times slower than the source server at Reading, on average. With Writing, the aggregating server was 12.44 times slower on average, while HistoryRead was only 1.58 times slower. However, HistoryRead was also nearly ten times slower than either Read or Write on the source server itself. Therefore, even a slight proportional slowdown results in a delay of several seconds, as the HistoryRead service is already slow.

Reading and writing on the aggregating server seem fast enough for most uses. A delay of 20-40 ms is not a major issue, unless the aggregating server is expected to provide near real time responses. When reading or writing historical data, a delay of 2-3 seconds should also be acceptable, as the history data generally does not change in the meantime. In the tested scenario with one aggregating server, the performance is sufficient. However, if the aggregating server itself would be further aggregated, the performance of the higher server would be even slower. Thus, the aggregating server prototype could use further optimization if it were to be used in OPC UA server networks with multiple layers of aggregation.

Overall, the testing results in this thesis show that the aggregating server prototype has room for improvement. The transformation slowing down as source servers increase in size and the outright failures to properly configure very large servers shows that the configuration process is insufficient for practical applications, where source server sizes can vary greatly.

7.2 Further research

Aggregating servers are reliant on subscriptions for presenting up-to-date data and evaluating the performance of subscriptions would be worth further study. For example, how quickly a subscription relays a value change to an aggregating server condition reliant on it determines how quickly the condition can change its state to the correct one and inform its client. One way to test subscription performance is by changing a source server value and measuring how long it takes for a corresponding aggregating server value to be notified of the change.

A more thorough testing and evaluation of time usage during service calls would allow identifying which parts of the process are the slowest. Improvement and optimization efforts could then be focused on these. For instance, it would be useful to know how time-consuming it is to find the original source node when relaying service calls.

Not designed in this work were the two other A&C cases, local conditions and a hybrid solution combining both relayed and local conditions. The difficulty with local conditions is adding a functionality to the transformation algorithm that adds conditions to some transformed nodes based on some given criteria. There are many possible situations where a condition could be needed, so a full rule system for the criteria would be complicated to create.

The source server address spaces in this work were static: No nodes were added or removed during use. The aggregating server prototype and its predecessor [8] do

not address the issue of what happens if nodes were to be created or removed on aggregated source servers. This is a possible use case for aggregating servers and should be examined.

Configuring the aggregating server prototype is currently fairly slow and does not scale well as the number of nodes on the source servers increases. A more optimized browsing method should be looked for. In addition, the transformation process currently only supports one-to-one transformations: Each node on the aggregating server can only have one or zero corresponding source server nodes, and vice versa. A many-to-one transformation process would allow features such as a combined condition that triggers if any of its many corresponding source server conditions trigger, or simple sums or averages of values from different source servers. Implementing many-to-one transformations offers multiple different options and should be examined carefully.

References

[1] Thilo Sauter and Maksim Lobashov. How to access factory floor information using internet technologies and gateways. Industrial Informatics, IEEE Transactions on, vol. 7, pages. 699–712, November 2011.

[2] OPC Foundation. OPC UA specification part 1: Overview and Concepts. Technical report, OPC Foundation, 2015. OPC UA Specification Release 1.03.

[3] Wolfgang Mahnke, Stefan-Helmut Leitner, and Matthias Damm. OPC Unified Architecture. Springer Publishing Company, Incorporated, 1st edition, 2009.

[4] Daniel Großmann, Markus Bregulla, Suprateek Banerjee, Dirk Schulz, and Roland Braun. OPC UA server aggregation - the foundation for an internet of portals. In Antoni Grau and Herminio Martínez, editors, Proceedings of the 2014 IEEE Emerging Technology and Factory Automation, ETFA 2014, Barcelona, Spain, September 16-19, 2014, pages 1–6. IEEE, 2014. OPC Foundation.

[5] OPC Foundation. OPC UA specification part 9: Alarms & Conditions. Technical report, OPC Foundation, 2015. OPC UA Specification Release 1.03.

[6] OPC Foundation. OPC UA specification part 11: Historical Access. Technical report, OPC Foundation, 2015. OPC UA Specification Release 1.03.

[7] Pyry Piirainen. OPC UA based remote access to agricultural field machines. Master's thesis, Aalto University School of Electrical Engineering, 2014.

[8] Tomi Tuovinen. OPC UA Address Space Transformations. Master's Thesis, Aalto University School of Electrical Engineering, 2015.

[9] Prosys OPC. https://www.prosysopc.com/. Accessed: 26.1.2016.

[10] Prosys OPC UA Historian. https://www.prosysopc.com/products/opc-ua-historian/. Accessed: 11.2.2016

[11] LibreOffice. https://www.libreoffice.org/. Accessed: 28.4.2017

[12] Unified Modeling Language. http://www.uml.org/. Accessed: 28.4.2017

[13] MatrikonOPC. http://www.matrikonopc.com/. Accessed: 26.1.2016.

[14] Softing. http://industrial.softing.com/en/. Accessed: 26.1.2016.

[15] Unified Automation. https://www.unified-automation.com/. Accessed: 26.1.2016.

[16] OPC Foundation. OPC UA specification part 3: Address space model. Technical report, OPC Foundation, 2015. OPC UA Specification Release 1.03.

[17] OPC Foundation. OPC UA specification part 13: Aggregates. Technical report, OPC Foundation, 2015. OPC UA Specification Release 1.03.

[18] OPC Foundation. OPC UA specification part 10: Programs. Technical report, OPC Foundation, 2015. OPC UA Specification Release 1.03.

[19] OPC Exhange: 6. Where do OPC timestamps come from? MatrikonOPC. http://blog.matrikonopc.com/index.php/ask-the-experts-opc-questions-and-answers/6-where-do-opc-timestamps-they-come-from/ . Accessed: 28.11.2016

[20] OPC Foundation. OPC UA specification part 4: Services. Technical report, OPC Foundation, 2015. OPC UA Specification Release 1.03.

[21] OPC Foundation. OPC UA specification part 8: Data Access. Technicalreport, OPC Foundation, 2015. OPC UA Specification Release 1.03.

[22] OPC Foundation. OPC UA specification part 5: Information Model. Technical report, OPC Foundation, 2015. OPC UA Specification Release 1.03.

[23] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In 2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA, Anaheim, CA, USA, 2003.

[24] Joona Elovaara. Aggregating opc ua server for remote access to agricultural work machines. Master's thesis, Aalto University School of Electrical Engineering, 2015.

[25] Jukka Asikainen. OPC UA Java History Gateway with Inherent Database Integration. Master's Thesis, Aalto University School of Electrical Engineering, 2013.